Code Density Concerns for New Architectures

Vincent M. Weaver
Cornell University
vince@csl.cornell.edu

Sally A. McKee
Chalmers University of Technology
mckee@chalmers.se

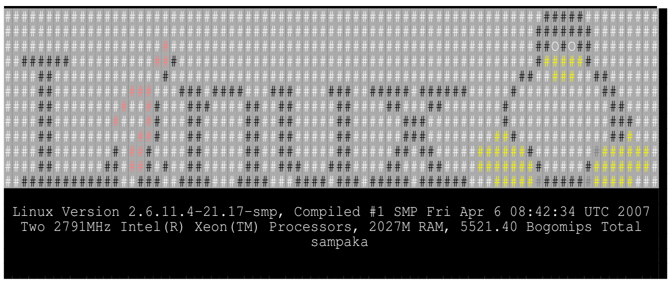
***Abstract*—**
Reducing a program's instruction count can improve cache behavior and bandwidth utilization, lower power consumption, and increase overall performance. Nonetheless, code density is an often overlooked feature in studying processor architectures. We hand-optimize an assembly language embedded benchmark for size on 21 different instruction set architectures, finding up to a factor of three difference in code sizes from ISA alone. We find that the architectural features that contribute most heavily to code density are instruction length, number of registers, availability of a zero register, bit-width, hardware divide units, number of instruction operands, and the availability of unaligned loads and stores.

We extend our results to investigate operating system, compiler, and system library effects on code density. We find that the executable starting address, executable format, and system call interface all affect program size. While ISA effects are important, the efficiency of the entire system stack must be taken into account when developing a new dense instruction set architecture.

I. Benefits of Code Density

Dense code yields many benefits. The L1 instruction cache can hold more instructions, which usually results in fewer cache misses [1]. Less bandwidth is required to fetch instructions from memory and disk [2], and less storage is needed to hold program images. With fewer instructions, more data fits in a combined L2 cache. Also, on modern multi-threaded processors, multiple threads share limited L1 cache space, so having fewer instructions can be advantageous. Denser code causes fewer TLB misses, since the code requires fewer virtual memory pages. Modern Intel processors, for instance, can execute compact loops entirely from the instruction buffer, removing the need for L1 I-cache accesses. Finally, the ability to consistently generate denser code can conserve power, since it enables smaller microarchitectural structures and uses less bandwidth [3], [4], [5], [6], [7].

Obviously, these benefits can come at a cost. For example, a denser ISA might require larger (and thus slower) pipeline decode stages, more complicated compilers, smaller logical register set sizes (due to limitations in the number of bits available in instructions), or even slower and more complex functional units. Compilers tend to optimize for performance, not size (even though the two are inextricably related): obtaining optimal code density often requires hand-tuned assembly language, which represents yet another tradeoff in terms of programmer time and maintainability. The current push for using CISC chips in the embedded market [8] forces a re-evaluation of existing ISAs.



```
Linux Version 2.6.11.4-21.17-smp, Compiled #1 SMP Fri Apr 6 08:42:34 UTC 2007
 Two 2791MHz Intel(R) Xeon(TM) Processors, 2027M RAM, 5521.40 Bogomips Total
                                   sampaka
```

Fig. 1. Sample output from the `linux_logo` benchmark

II. Methodology

Investigations of code density often use microbenchmarks (which tend to be short and not representative of actual workloads) or else industry standard benchmarks (which are written in high-level languages and thus are limited by compiler code generation capabilities). As a compromise, we take an actual system utility, but convert it into pure assembly language in order to directly interact with the underlying ISA. We hand-optimize it for size, attempting to create the smallest binary possible, even if this potentially creates slower code. The program we choose, `linux_logo` [9], is a utility available with many Linux distributions. When given a sufficiently large input set, its characteristics are similar to the `stringsearch` benchmark included with the MiBench [10] suite. The program executes various syscalls to gather system information, then displays this info along with a colorful ASCII penguin (Figure 1 shows sample output).

The stock `linux_logo` program contains a multitude of features and command line options; we remove all but the minimum for simplicity. Remaining code is divided into two parts: the first decodes and displays the text logo, which is packed using LZSS compression [11], [12]; the second prints system information, which is gathered by reading the Linux `/proc/cpuinfo` file, in addition to invoking the `uname()` and `sysinfo()` syscalls. Major subroutines include string copying, string searching, integer to ASCII conversion, and centering routines. The code makes system calls directly to avoid C library overheads. Code is assembled with the GNU assembler and is linked with GNU ld. Executables are stripped of non-essential data using the `sstrip` "super strip" program [13], an enhanced version of the UNIX strip command. Executables are tested on actual hardware or under an emulator where hardware is unavailable.

We attempt to optimize each architecture's code to the minimum possible size without corrupting correct results.

TABLE I
SUMMARY OF INVESTIGATED ARCHITECTURES

Type	arch	endian*	bits	instr len (bytes)	op args	GP int regs	unaligned ld/st	auto-inc address	hw div	stat flags	branch delay	predi-cation
VLIW	IA64	little	64	16/3†	3	127,zero	no	yes	no	yes	no	yes
RISC	Alpha	little	64	4	3	31, zero	no	no	no	no	no	no
	ARM	little	32	4	3	15,PC	no	yes	no	yes	no	yes
	m88k	big	32	4	3	31,zero	no	no	Q only	no	optional	no
	MicroBlaze	big	32	4	3	31,zero	no	no	Q only**	no	optional	no
	MIPS	big	32/64	4	3	31,hi/lo,zero	yes**	no	yes	no	yes	no
	PA-RISC	big	32/64	4	3	31,zero	no	no	part	no	yes	no
	PPC	big	32/64	4	3	32	yes	yes	Q only	yes	no	no
	SPARC	big	32/64	4	3	63-527,zero‡	no	no	Q only	yes	yes	no
CISC	m68k	big	32	2-22	2	16	yes	yes	yes	yes	no	no
	s390	big	32/64	2-6	2	16	yes	no	yes	yes	no	no
	VAX	big	32	1-54	3	16	yes	yes	yes	yes	no	no
	x86	little	32	1-15	2	8	yes	yes	yes	yes	no	no
	x86_64	little	32/64	1-15	2	16	yes	yes	yes	yes	no	no
Embedded	AVR32	big	32	2	2	15,PC	yes	yes	yes	yes	no	no
	CRISv32	little	32	2-6	2	16,zero,special	yes	yes	part	yes	yes	no
	SH3	little	32	2	2	16,MAC	no	yes	part	yes	yes	no
	THUMB	little	32	2	2	8/15,PC	no	yes	no	yes	no	no
8/16-bit	6502	little	8	1-3	1	3	yes	no	no	yes	no	no
	PDP-11	little	16	2-6	2	6,sp,pc	no	yes	yes**	yes	no	no
	z80	little	8	1-4	2	18	no	lim	no	yes	no	no

* on the machine we used † 16-byte bundle has 3 instructions ‡ register windows, only 32 visible ** many implementations

For RISC architectures with fixed-length instructions this is easier: typically, there is only one way to express an operation, so there are limitations to clever implementations. Optimizations are limited to trying to load 32-bit constants in a small area, using registers instead of memory, and using tail merging to shorten procedure lengths. CISC architectures provide many more opportunities to decrease code size, but it is much more difficult to track optimizations due to variable-length instructions. Optimizing for density requires frequent disassembler checks to verify sizes of individual instructions. Interestingly, we find that the "do-everything" super-CISC instructions available on these systems can often be implemented with a smaller set of simpler CISC instructions.

III. ARCHITECTURAL NOTES

Table I lists relevant features of the architectures of interest. We present a broad overview of these architectures.

VLIW: Very Long Instruction Word (VLIW) architectures are designed to take advantage of parallelism in code. If the code is not inherently parallel (and ours is not), code density suffers, and many operations are wasted as nops. Writing compact VLIW code can be hard: resolving dependences correctly is a difficult task for compilers, and an even more difficult task for programmers writing assembly by hand. VLIW can be designed with code density in mind: e.g., the WM [14] architecture could exploit two operations per instruction in over two-thirds of all cases. The only VLIW architecture we investigate is Intel's IA64 [15].

RISC: Reduced Instruction Set Computers (RISC) emphasize simple architectures with easy to decode instructions. Instruction length is fixed at four bytes, which necessitates inefficiency in instruction encoding. These are load-store architectures, which require moving memory values into registers before operating on them (this negatively impacts code density). Some of these architectures stretch the definition of "reduced"; the PowerPC architecture has nine different add instructions, and has the `rlwimi` (rotate left word immediate then mask insert) instruction, which takes five parameters. We investigate the Alpha [16], ARM [17], m88k [18], MicroBlaze [19], MIPS [20], PA-RISC [21], PowerPC [22], and SPARC [23] ISAs.

CISC: Complex Instruction Set Computers (CISC) tend to have high code density. Most CISC architectures have variable-sized instructions, which makes processor decode more complicated, but allows for dense code. An example of a dense "complex" instruction is the x86 one-byte `lodsb` instruction, which both loads a byte from memory and increments a pointer. Another impressively complex instruction is the VAX `matchc`, which does a full "find substring x inside of string y in memory." Compilers often have difficulty using these instructions appropriately, so this potential for density can be wasted. Also, these instructions may not be shorter or faster than a set of simpler instructions performing the same operations. We investigate the m68k [24], s390 [25], VAX [26], x86 [27], and AMD64 [28] ISAs.

Embedded: Modern advances in CPU design have pushed the limits of what qualifies as "embedded". We use the term to refer to any architecture with a fixed two-byte instruction length, but capable of running a modern 32-bit Linux kernel. These processors tend to have consistently small code sizes, but can still be beaten by variable-instruction length CISC systems. We investigate the AVR32 [29], CRISv32 [30], SH3 [31], and ARM THUMB [17] ISAs.

8 and 16 bit: For comparison purposes we investigate older processors with smaller word sizes. Such CPUs are still used for embedded systems, and they are designed for use where code density is a much more critical concern. We investigate the 6502 [32], PDP-11 [33], and z80 [34] ISAs.

TABLE II
CORRELATIONS OF ARCHITECTURAL FEATURES TO BINARY SIZE

Correlation Coefficient	Architectural Parameter
0.9381	Minimum possible instruction length
0.9116	Number of integer registers
0.7823	Virtual address of first instruction
0.6607	Architecture has a zero register
0.6159	Bit-width
0.4982	Number of operands in each instruction
0.3129	Year the architecture was introduced
-0.0021	Branch delay slot
-0.0809	Machine is big-endian
-0.2121	Auto-incrementing addressing scheme
-0.2521	Hardware status flags (zero/overflow/etc.)
-0.3653	Unaligned load/store available
-0.3854	Hardware divide in ALU

IV. Code Density Findings

Table II shows how architectural features contribute to code size. A positive correlation means that high values of the feature increase code size; a negative correlation means that high values decrease code size. Figure 2 shows total binary sizes across the investigated architectures and Figures 3, 4, 5, and 6 show code sizes of various components. We detail causes of these trends below.

Minimum instruction length: Short instruction encodings help most with respect to reducing density. Architectures with variable-length instructions, especially those with useful single-byte instructions (like x86 and VAX), can accomplish much work with little code. Fixed-length ISAs can be dense if all instructions are 16-bit (like AVR32 and SH3); RISCs with fixed 32-bit instructions generate less dense code; and the VLIW generates the least dense code of all platforms studied. Figure 3's LZSS decompression clearly demonstrates this.

Number of integer registers: Having fewer registers reduces the number of bits needed to encode instructions, increasing code density. There is a tradeoff, in that having fewer registers generates more loads/stores from spilling in load-store architectures.

Virtual address of first instruction: Operating system design decisions affect code density. If the virtual address space is configured so programs start near the bottom of virtual memory, then a 16-bit constant is enough to point to a small program's entire memory. Constant 32-bit pointer loads are at least double the size of 16-bit loads on most architectures, and 64-bit pointer loads are even more wasteful. Using small system call numbers can help, too; avoiding large immediate constants saves space in executables.

Existence of a zero register: Zero registers are normally found in RISC architectures, so they tend to correlate with less dense code. A zero register can be simulated using one load instruction and sacrificing a register, so the feature offers few benefits with regards to code density.

Bit width: Having a narrower bit-width leads to denser code, mainly due to shorter immediate values for pointer loads and branch offsets.

Number of operations in instruction: Operation count directly affects the size of instruction encoding.

Year of introduction: Somewhat surprisingly, age does not correlate highly with code density. This is due to the many embedded architectures introduced recently.

Branch delay slots: Branch delay slots can decrease code density due to added nops. For our benchmark, slots can often be filled, so branch delay slots cause no problem.

Endianess: Endianess has little impact on code density unless the program operates on data in a non-native format.

Status flags: Upon completion of ALU operations, these flags (or condition codes) are set as side effects to indicate that the result was zero, negative, an overflow, etc. These flags can lead to denser code by eliminating the need for comparison instructions before conditional branches. Most RISC designs avoid status flags, as they add complexity and ordering dependencies to out-of-order processors.

Auto-increment addressing: Auto-increment addressing modes allow accessing consecutive memory addresses without requiring separate increment instructions. This is especially useful for accessing arrays, of which C strings are a subset. String copying and concatenation, as in Figure 4, benefit from these instructions.

Unaligned memory access: Allowing unaligned loads and stores leads to smaller code, especially for string manipulation. Unaligned 16 and 32 bit loads permit arbitrary simultaneous access to consecutive bytes in memory. If alignment is enforced, achieving the same results requires a series of memory, shift, and logical operations. Results in Figure 5 demonstrate benefits of this feature.

Hardware division: A hardware divide instruction is often slower than using the equivalent multiply by the reciprocal [35] or lookup table-based division routines, but it almost always takes fewer bytes in the instruction stream. Some architectures only implement single-bit division routines that require software pipelining; this can lead to less space-efficient code than otherwise undesirable algorithms such as iterative subtraction. Integer printing code benefits greatly from hardware divide, as in Figure 6.

V. Density of Compiler-Generated Binaries

Hand-optimizing large programs in assembly language is impractical under most circumstances. We therefore evaluate compact code generation using more traditional methods. We choose to experiment with the x86 architecture due to its popularity and high code density.

We use a variety of C compilers and libraries to determine how small an executable we can generate using off-the-shelf tools. We use the GNU gcc 4.2 compiler (gcc 4.1 for uClibc runs), the Intel C compiler version 9.1.038, and the SunStudio 12 compiler, all under Linux. We use GNU libc 2.7 and the embedded uClibc 0.9.27.

We experiment with different compiler optimizations. In general, we use `-O3`; this usually optimizes for maximum performance. We also evaluate `-Os`, which optimizes for size. In practice, resulting executables are very similar. The primary differences are lack of loop unrolling, use of the

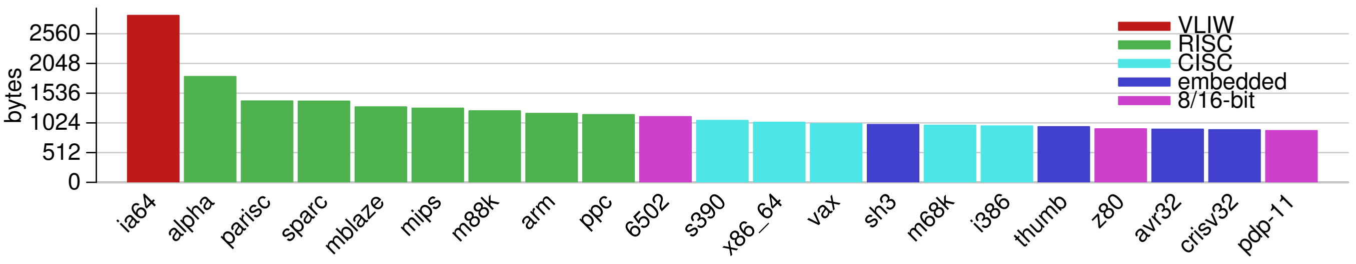


Fig. 2. Total size of benchmarks (includes some platform-specific code, so does not strictly reflect code density)

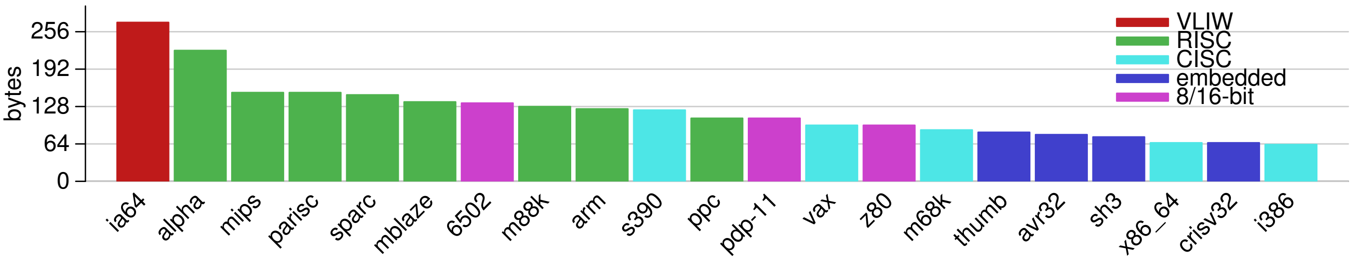


Fig. 3. Size of LZSS decompression code

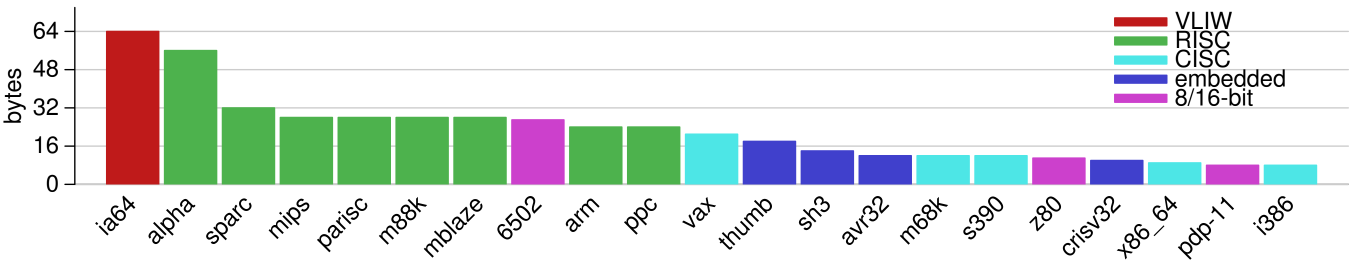


Fig. 4. Size of string concatenation code (machines with auto-increment addressing modes and dedicated string instructions perform better)

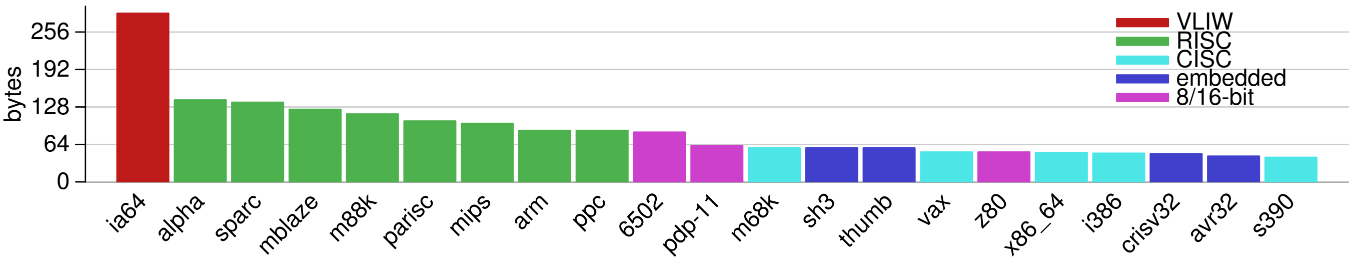


Fig. 5. Size of string searching code (unaligned load instructions help, since four bytes at arbitrary offsets can be compared at once. CISC architectures as well as avr32 and MIPS benefit)

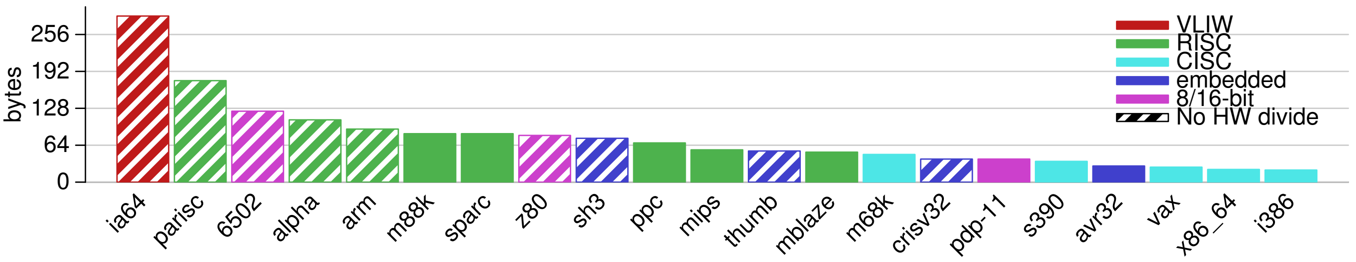


Fig. 6. Size of integer printing code (hardware divide helps code density)

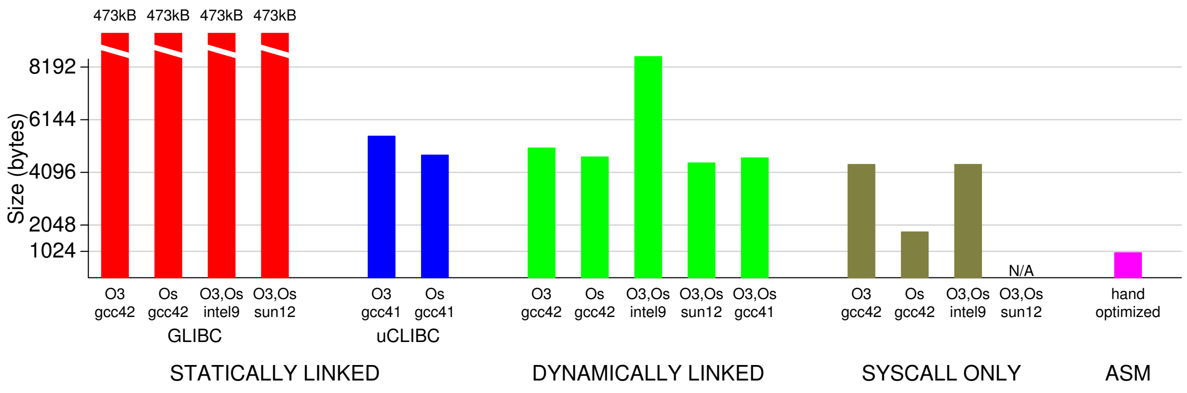


Fig. 7. Total size of generated executables, stripped of debugging information.

hardware divide instruction instead of the faster multiply by reciprocal method, lack of function inlining, and less aggressive padding of function entry points.

Figure 7 shows that executable sizes vary by many orders of magnitude. This is because statically linked programs contain the entire C library, which represents an overhead of at least 450KB (when using glibc).

By writing code that avoids the C library (and using system calls directly), we obtain executables only twice as large as hand-optimized codes. The remaining reasons for larger code are:

- setting up the stack frame pointer at function entry — this can be turned off with the compiler option `---fomit-frame-pointer`;
- writing back to memory using 32-bit constants — due to pointer aliasing issues the compiler must frequently write values to memory using 5-byte instructions. The optimized assembler avoids aliasing and places more values in registers;
- loading of constants inefficiently — there are various slow (but smaller) ways to load small constants on x86; and
- avoiding string instructions — the compiler simply does not use the x86 specialized string instructions.

VI. Related Work

Most code density research addresses the compressibility of instruction code [36], [37], [38], [39], [40], [41], [42], [4], [43], [44], [45]. Usually what is compressed is compiler-generated RISC or VLIW code, with compression ratios typically in the 50-70% range. We show here that embedded and CISC ISAs yield smaller binaries than RISC. Adding compression to a RISC architecture likely negates the speed benefits and decoder simplicity that initially motivated the move away from CISC.

Previous work compares multiple architectures, but our work is unique in the number (21) considered. Kozuch and Wolfe [46] measure entropy and compressibility of six different architectures (VAX, MIPS, SPARC, m68k, RS6000 and PowerPC). Hasegawa et al. [3] compare SH3 code density to that of code generated by gcc on 10 other platforms (m68k, IA32, i960, Sparclite, SPARC, MIPS, AMD29k, m88k, Alpha, and RS6000). They find results roughly similar to ours, though they find the SH3 architecture generates smaller code than the x86 and m68k by a small margin. Flynn, Mitchell, and Mulder [47] compare code density of synthetic architectures that do not model actual systems.

Phelan [48] investigates features added to Thumb-2 to increase code density. Thumb-2 uses specialized instructions for enhanced constant support, limited predication, and compare-against-zero. These are similar features to those we find useful for density in Section IV. Halambi et al. [49] investigate the benefits of using a *reduced Instruction Set Architecture* (rISA), such as THUMB and MIPS-16. They test hypothetical architectures, finding that a hybrid approach unlike any current reduced architecture should perform best.

Massalin's Superoptimizer [50] cleverly generates extremely dense (and non-intuitive) m68k and IA32 code by exhaustive search, but it only operates on small blocks of code (i.e., it's a highly tuned peephole optimizer).

VII. Conclusions and Future Work

A 1987 article by Chow and Horowitz [51] quotes an early MIPS-X design document:

> "The goal of any instruction format should be: 1. simple decode, 2. simple decode, and 3. simple decode. Any attempts at improved code density at the expense of CPU performance should be ridiculed at every opportunity."

Two decades later, the debate between prioritizing code density versus decoder simplicity in ISAs continues.

We investigate code density of 21 different architectures, and find that very high density levels can be achieved with proper planning of an ISA. To thoroughly exploit ISA density there must be cooperation between the operating system, system libraries, and compiler. On the x86 architecture, even after eliminating the C library and choosing maximum compiler options, a factor of two in code density can still be realized by hand-optimizing the assembly code. This is much greater than the 25% average size difference between RISC and CISC codes.

New ISAs, especially embedded ones, are continually being developed. Now that FPGAs are powerful enough to contain competitive CPUs, this trend of creating custom ISAs will likely increase. To aid in this development, we show which architectural features contribute most to code density, but also show that the entire system stack must be optimized to avoid wasting an ISA's inherent potential for density.

Ongoing work applies some of what we have learned to much bigger benchmarks to see what the performance and power implications are of using smaller libraries and different compiler options on larger applications. We hope to raise awareness of the importance of code density on all modern architectures, not just those targeted for the embedded space.

References

[1] P. Steenkiste, "The impact of code density on instruction cache performance," in *Proc. 16th IEEE/ACM International Symposium on Computer Architecture*, June 1989, pp. 252–259.

[2] J. Davidson and R. Vaughan, "The effect of instruction set complexity on program size and memory performance," in *Proc. 2nd ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1987, pp. 60–64.

[3] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas, "SH3: High code density, low power," *IEEE Micro*, vol. 15, no. 6, pp. 11–19, 1995.

[4] E. Wanderley Netto, R. Azevedo, P. Centoducatte, and G. Araujo, "Multi-profile based code compression," in *Proc. 41st ACM/IEEE Design Automation Conference*, June 2004, pp. 244–249.

[5] A. Zmily and C. Kozyrakis, "Simultaneously improving code size, performance, and energy in embedded processors," in *Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition*, Mar. 2006, pp. 224–229.

[6] L. Benini, A. Macii, and A. Nannarelli, "Cached-code compression for energy minimization in embedded processors," in *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, Aug. 2001, pp. 322–327.

[7] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proc. IEEE/ACM 33nd International Symposium on Microarchitecture*, Dec. 2000, pp. 245–257.

[8] B. Smith, "ARM and Intel battle over the mobile chip's future," *IEEE Computer*, May 2008.

[9] V. Weaver, http://www.deater.net/weave/vmwprod/linux_logo/, 2009.

[10] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE 4th Workshop on Workload Characterization*, Dec. 2001, pp. 3–14.

[11] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[12] J. Storer and T. Szymanski, "Data compression via textual substitution," *Journal of the ACM*, vol. 29, pp. 928–951, 1982.

[13] B. Raiter, http://www.muppetlabs.com/~breadbox/software/elfkickers.html, 2007.

[14] W. Wulf, "Evaluation of the WM architecture," in *Proc. 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 382–390.

[15] *Intel Itanium Architecture Software Developer's Manual*, Intel, 2000.

[16] *Alpha Architecture Handbook*, Compaq Computer Corporation, 1998.

[17] *ARM Architecture Reference Manual*, ARM Limited, 2000.

[18] *Motorola MC88110 User's Manual*, Motorola, Inc., 1991.

[19] *MicroBlaze Processor Reference Guide*, Xilinx, 2004.

[20] *MIPS32 Architecture for Programmers*, MIPS Technologies, Inc., 2001.

[21] *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, Hewlett Packard, 1994.

[22] *PowerPC Microprocessor Family: The Programming Environments for 32-bit Microprocessors*, IBM, 2000.

[23] *The SPARC Architecture Manual Version 9*, Sun Microsystems, 1994.

[24] *Motorola M68000 Family Programmer's Reference Manual*, Motorola, Inc., 1992.

[25] *Enterprise Systems Architecture/390: Principles of Operation*, IBM, 1999.

[26] *VAX Architecture Reference Manual*, Digital Equipment Corp., 1987.

[27] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corp., 2007.

[28] *AMD64 Architecture Programmer's Manual*, Advanced Micro Devices, 2006.

[29] *AVR32 Architecture Document*, Atmel, 2006.

[30] *ETRAX FS Designer's Reference*, Axis Communications AB, 2007.

[31] *SH-3/SH-3E/SH3-DSP Software Manual*, Renesas Technology, 2006.

[32] *MCS6500 Microcomputer Family Hardware Manual*, MOS Technology Inc., 1975.

[33] *pdp11/40 Processor Handbook*, Digital Equipment Corp., 1972.

[34] *Z80 family CPU User Manual*, Zilog, 2001.

[35] T. Granlund and L. Montgomery, "Division by invariant integers using multiplication," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994, pp. 61–72.

[36] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *Proc. IEEE/ACM 25th International Symposium on Microarchitecture*, Nov. 1992, pp. 81–91.

[37] H. Lekatsas and W. Wolf, "SAMC: A code compression algorithm for embedded processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 12, pp. 1689–1701, 1999.

[38] B. De Sutter, B. De Bus, K. De Bosschere, and S. Debray, "Combining global code and data compaction," in *Proc. of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 2001, pp. 29–38.

[39] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto, "Survey of code-size reduction methods," *ACM Computing Surveys*, vol. 35, no. 3, pp. 223–267, Sept. 2003.

[40] J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder, "Reducing code size with echo instructions," in *Proc. 7th ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Oct. 2003, pp. 84–94.

[41] X. Xu, C. Clarke, and S. Jones, "High performance code compression architecture for the embedded ARM/THUMB processor," in *Proc. ACM Computing Frontiers Conference*, Apr. 2004, pp. 451–456.

[42] C. Lin, Y. Xie, and W. Wolf, "LZW-based code compression for VLIW embedded systems," in *Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition*, Feb. 2004, pp. 76–81.

[43] Y. Wu, M. Breternitz, Jr., H. Hum, R. Peri, and J. Pickett, "Enhanced code density of embedded CISC processors with echo technology," in *Proc. 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Oct. 2005, pp. 160–165.

[44] S. Seong and P. Mishra, "A bitmask-based code compression technique for embedded systems," in *Proc. International Conference on Computer Aided Design*, Nov. 2006, pp. 251–254.

[45] T. Bonny and J. Henkel, "Efficient code density through look-up table compression," in *Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition*, Apr. 2007, pp. 809–814.

[46] M. Kozuch and A. Wolfe, "Compression of embedded system programs," in *Proc. IEEE International Conference on Computer Design*, Oct. 1994, pp. 270–277.

[47] M. Flynn, C. Mitchell, and J. Mulder, "And now a case for more complex instruction sets," *IEEE Computer*, vol. 20, no. 9, pp. 71–83, 1987.

[48] R. Phelan, *Improving ARM Code Density and Performance: New Thumb Extensions to the ARM Architecture*, ARM Limited, 2003.

[49] A. Halambi, A. Shrivastava, P. Biswas, N. Dutt, and A. Nicolau, "A design space exploration framework for reduced bit-width instruction set architecture (rISA) design," in *Proc. 15th IEEE/ACM International Symposium on System Synthesis*, Nov. 2002, pp. 120–125.

[50] H. Massalin, "Superoptimizer: a look at the smallest program," in *Proc. 2nd ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1987, pp. 122–126.

[51] P. Chow and M. Horowitz, "Architectural tradeoffs in the design of MIPS-X," in *Proc. 14th IEEE/ACM International Symposium on Computer Architecture*, June 1987, pp. 300–308.